View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Computing and Informatics (E-Journal - Institute of Informatics, SAS,...

Computing and Informatics, Vol. 31, 2012, 983–1002

ASPECT-ORIENTED APPROACH TO METAMODEL ABSTRACTION

Ján Kollár

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
e-mail: Jan.Kollar@tuke.sk

Michal Vagač

Department of Informatics
Faculty of Natural Sciences
Matej Bel University
Tajovského 40, 974 01 Banská Bystrica, Slovakia
e-mail: michal.vagac@gmail.com

Communicated by Ivan Luković

Abstract. A software system maintenance represents an important part of software system's lifetime. The most common reasons to change a software system are bug fixes and adding of a new functionality. Software maintenance itself is a difficult and complex process. Before applying a change, it is important to understand the software system's source code as well as the application domain. This paper presents our innovative approach to improve software system comprehension in order to simplify its maintenance. Instead of analyzing all the program code, our approach focuses on parts which are built using predefined well known software libraries. The knowledge of both – the libraries and the way they are used in software systems – allows us to identify certain concepts of the software system. This information is used to create metamodels of these concepts. The metamodel is created at a higher level of abstraction than the level of concept implementation.

Keywords: Program comprehension, software adaptation, metalevel architecture, metaprogramming, aspect-oriented programming

Mathematics Subject Classification 2010: 68N15, 68N20

1 INTRODUCTION

Computers (and software systems they run) help us with our everyday tasks. No software system is perfect and sooner or later there is a requirement for a change. Often, software systems model different problems of the real world. Continuous changes in this world lead to a state in which the software system is becoming insufficient. In that moment it is required to reflect these real world changes also in the software system. Another reason for a change may be different constraints during initial system development, or changes in order to constrain system's inner complexity growth [18, 21]. Because of all these reasons, a software system faces many changes during its lifetime – in general most common reasons are bug fixes and adding a new functionality. Software evolution represents a need for continuing maintenance and development of software used in real world applications or to solve problems in a real world domain [19]. It is more common to change the existing system than to create a new one. Software system maintenance and evolution consumes up to 80 percent of its lifetime [20].

The topic about applying a software change became an important field of research. Software maintenance itself is a difficult and complex process. An important area related to applying a software change is correct and effective program comprehension. Before applying a change, it is important to understand the software system's source code as well as the application domain. Only after a full understanding of a program (or at least the affected submodule) it is possible to apply the required change. Without sufficient knowledge it is easily possible to break the functionality of the system. Computer program is subject not only to a computer, but also to its developer – a programmer. To change a program, the programmer must read it and understand it. Improvements in ways of program comprehension affect program maintenance as well.

1.1 Program Comprehension

One of the first attempts to describe a theory about program understanding was presented in [5]. Program understanding is described as a succession of knowledge domains that bridge the problem being solved and the executing program. As examples of knowledge domains there are the application (or problem) domain, the algorithm domain, the domain of translation of the algorithm into a programming language, or the domain of execution of the program. One side of this succession works with human-oriented terms (designed for human level communication), while the other side uses vocabulary and grammar which are narrowly restricted and formally controlled (designed for automated treatment) [2]. The programming process is a construction of mappings from the application domain, through one or more

intermediate domains, to the implementation domain. The comprehension process is the reconstruction of all or part of those mappings [35].

To understand a program, a programmer uses internal (comments, names in code) and external (documentation) sources of information to successively refine hypotheses about the program's operation. These hypotheses are initially generated from the programmer's knowledge of the task domain and of programming. A detailed analysis of the comprehension process is presented in [29]. As a programmer reads a program code, he/she learns more about the program – this new knowledge can be then reused in the following searches.

The paper [6] identified four essential properties of a software system which make building software system a difficult task. All of these properties also influence difficulty of program comprehension.

Complexity – software entities are one of the most complex human products. No two parts are alike (if they are, we make two similar parts into one).

Conformity – much complexity comes from conformation to other interfaces; this cannot be simplified only by redesigning of the software.

Changeability – the software entity is constantly subject to pressures for a change.

Invisibility – the software is invisible. To visualize the software several different diagrams are used to display different properties of the software.

The method proposed in this paper deals with complexity and invisibility difficulties.

The work [30] describes comprehension by a triangular diagram (Figure 1). The triangle has the vertices *name*, *intension* (concept, feature) and *extension* (implementation, plan). Intension can be seen as a simple general description, while extension can be seen as a concrete instance. The process of comprehension consists in possibility of moving among all three vertices of the triangle. The triangle describes six fundamental comprehension processes:

naming – gives a name to an intension

definition – finds the corresponding intension for a given name

recognition – recognizes in an extension a corresponding intension

location – finds for a given intension a corresponding extension

annotation – recognizes an extension and gives it a name

traceability – finds corresponding extensions for a given name.

The process of identification of fragments of the program code related to known functionality of a program is known as concept (or feature) location [37]. A concept represents a well understood abstraction of a system's problem domain [34]. The task is to identify mappings between domain level concepts and their implementations in the source code [27]. The input of the mapping is the maintenance request, expressed usually in the natural language, using the domain level terminology. The output of the mapping is a set of components that implement the concept [8]. The input and output of location process belong to different levels of abstraction.

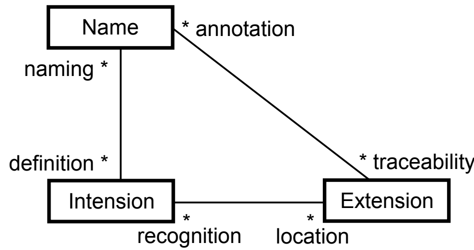

Fig. 1. Fundamental comprehension processes [30]

Before software system maintenance, a developer must localize parts of a program related to the change request. Therefore concept location is a prerequisite of code changes. The literature reports concept location approaches built on static analysis [2, 24, 41], dynamic analysis [37, 34], or their combination [22].

Our approach allows to automatically locate the predefined set of concepts in an unknown program. The located concept is presented at a higher level of abstraction than its implementation. It means the automation in mapping between several knowledge domains – the domains that the user would otherwise have to be familiar with.

In the approach described, the located concept is presented as a metamodel – one metamodel describes one located concept. Concept location and building of metamodels use properties of aspect-oriented programming. Basics about metalevel architectures and aspect-oriented programming are presented in next two subsections.

1.2 Metalevel Architectures

A metalevel architecture consists of different levels, where one level is controlled by another one. From a view of object-oriented programming, where a program is represented as a set of objects, it is possible to define several terms in the area of metalevel architectures. An application describing a problem being solved is located at the *domain level*. *Domain objects* are objects of this application. These objects describe the problem being solved. A *domain object protocol* defines operations provided by a domain object. A *domain operation* is an operation from the domain object protocol.

Beside the domain level there is a *metalevel*, which provides a space for metaobjects. *Metaobjects* describe, control, implement or modify domain objects. In case of a multilevel architecture, a metaobject can control another metaobjects. A *metaobject protocol* (MOP) is an object-oriented interface allowing communication between objects at the domain level and objects at the metalevel. It defines an application programming interface which can be used to work with metaobjects. Finally, a *metaobject operation* is an operation from a metaobject protocol.

Our approach is built as a metalevel architecture. The base level is represented by a software system (legacy application). The metalevel contains a metasystem which builds metamodels of located concepts.

1.3 Aspect-Oriented Programming

Aspect-oriented programming allowed modularization of crosscutting concerns. The *crosscutting concerns* are concerns which are impossible to modularize by available language constructs, such as classes or methods. Modularizing these concerns would break modularization of other described concerns. Two concerns crosscut when they have to be composed differently, but at the same time they must be coordinated [16]. A typical example of a concern that tends to be crosscutting is logging. Logging affects every logged part of the system – thus cannot be modularized in one single place. Crosscutting concerns result in a tangled code, which is difficult to understand and reuse.

An *aspect* describes *crosscutting concerns* – it allows their modularization, which results in a more simple code which is easier to maintain. *Join points* are points in which an aspect crosscuts the basic program. By defining a *pointcut*, it is possible to define a whole set of joint points. In the logging example, a pointcut could describe each method invocation. An *advice* allows to define an action executed at points defined by the pointcut (e.g. log information about method invocation). An aspect represents a modular unit consisting of a pointcut and an advice. Composing defined aspects and affected program is the task of the aspect weaver. The most common way of aspect implementation is weaving the aspect code into the program code.

Beside clearer modularization possibilities, aspect-oriented programming allowed adding a new functionality into an existing code. An aspect is defined in such a way that join points are defined to point at the place in the program which needs to be enhanced. An advice then contains an enhancement code.

The approach proposed uses this property of aspect-oriented programming. The base level software system is enhanced with a new code, which traces program execution and looks for implementation of predefined concepts. The same property of AOP is used later to apply a software system change.

The rest of the paper is organized as follows: in the following section we present our method of creating a metamodel at a higher level of abstraction than the implementation level and describe our experimental tool following this method. Then the work related to our approach is described. In the last section we present conclusions.

2 METAMODEL AT A HIGHER LEVEL OF ABSTRACTION

As presented in Section 1.1, a change request is expressed in the natural language using the domain level terminology. To implement a change, a developer must identify mappings between domain level concepts and their implementations in a source code. Domain level concepts and implementation belong to different levels of abstraction.

To find mappings between different levels of abstraction, a developer must have certain knowledge. The knowledge is gained from previous experiences and studies. A scheme of a tool with the capability of a program understanding was outlined in [30] or [36] (Figure 2).

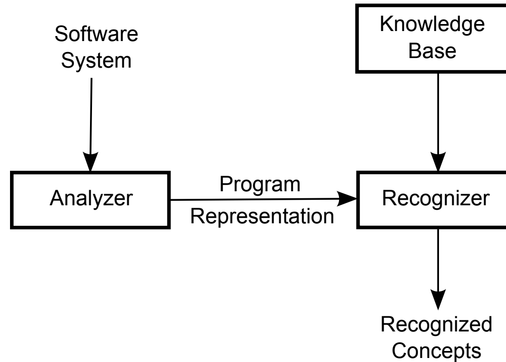

Fig. 2. A tool with capability of a program understanding

In the first step, the software system is analyzed. The analysis can be static (analysis of source code) or dynamic (analysis of a system execution). The output of the analyzer is a representation of the analyzed program. The representation can describe all system or only its part. In the next step, the recognizer – with the help of the knowledge base – recognizes familiar fragments in the analyzed system.

Source code analysis and execution traces analysis are commonly used procedures. Unclear parts from the depicted figure are the knowledge base and the recognizer. According to the information from the knowledge base, the recognizer must recognize fragments in the program representation. The software system is usually a very complex system – one problem can be solved in several different ways. To be able to recognize all possible concepts, the knowledge base must contain all these concepts pre-defined and their program representation (implementation).

When building a complex software system, a significant part of the implementation is repeatedly the same. To make the development process easier – without reinventing the wheel – the modern programming languages provide a standard library with reusable software components which implement many different general tasks.

Our approach focuses on the object-oriented paradigm. This paradigm is supported by the most widely used languages nowadays (Java, C++, C#, PHP). A program developed in an object-oriented language is typically defined by a group of classes and their instances – objects. Objects communicate with each other by sending messages. The structure of a class and relationships between classes and objects are specified by a program code.

To create a higher level of abstraction of the analyzed program automatically, it is needed to find control structures, analyze relationships between classes (objects)

and to understand meaning of the classes. First two points can be realized by different techniques of source code (or execution traces) analysis.

The situation is not so simple with the knowledge base. The knowledge base must contain the information needed to map acquired program representation into a recognized concept. As noted in the Introduction section, these two are situated at different levels of abstraction (implementation domain vs application domain).

There are many possibilities how to build a program in a general-purpose programming language, therefore it is hard (if possible) to prepare a knowledge base which would allow to recognize all possible concepts. In the approach proposed, we have focused on well-known parts represented by the software libraries used in the program. In a program written in a general-purpose programming language, we will search for parts which use standard libraries. The knowledge base will contain exact information about concepts implemented by a standard library. This information must include both – implementation details, concept details and the way of mapping between these two.

Concept implementation must be described in a very high detail. To be able to identify a concept from its implementation, there must be a knowledge about classes which implement the concept and in what relationships those classes can be. There is a possibility that the same class will be used to implement different concepts. The knowledge base must distinguish such concepts from the way of their implementation.

The next section describes architecture details of the proposed approach.

3 THE METAMODEL CREATION METHOD

The approach proposed is designed as a metalevel architecture. The base level of this architecture is represented by the software system we deal with (a concept implementation). The metalevel provides a method for analyzing and recognizing concepts implemented in the base level. The result of the analysis and recognition – a recognized concept – is presented as a metamodel. The metamodel is stored at the metalevel.

To build the proposed architecture there is a need to solve several challenges. According to Figure 2, the first step in the understanding process is a software system analysis. The result of the analysis is stored as a program representation, which serves as input to a recognition process. The recognition process uses the knowledge base and tries to recognize the program representation.

As the analyzer has no information about a concrete concept (it has no connection with the knowledge base), the analyzer must be general for all recognized concepts. Also the program representation must be common for all analyzed data. As a result, the program recognizer has to work with the same general data for all recognized concepts.

In the approach proposed, we suggest to integrate analyzer and recognizer subsystems into one part with accesses to the knowledge base (Figure 3). In this case,

instead of analyzing the whole program and recognizing general program representation, the analyzer can focus only on those parts of implementation, which implement predefined concepts. Since the analyzer has a direct access to the knowledge base, it has exact information about the concept implementation. Since analysis is focused on a certain concept implementation, the result of the analysis is a recognized concept. Thus, recognition is performed during analysis process.

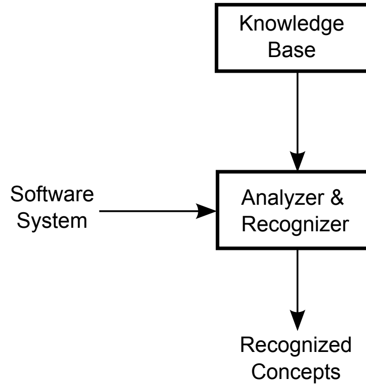

Fig. 3. Proposed tool with capability of a program understanding

The uniqueness of the class implementing the concept must be ensured. If an application will define classes with exactly the same name as those in the standard library (and defined in the knowledge base), the system will not be able to handle this situation properly – the recognition will not work.

3.1 The Base Level Monitoring Aspect

Aspect-oriented programming together with a better modularization allowed also extending the existing code with a new functionality [28]. It is even possible to add a new functionality without access to the source code. This property is very helpful for techniques depending on metadata as well – it is possible to extend the base level system with a monitoring code [28]. This new code will get required information which is used to build metamodel at the metalevel.

The mentioned properties of aspect-oriented programming were crucial for choosing this paradigm as a way of collecting information about the base level software system. With the help of aspect-oriented programming, the base level system is extended with a new code, which has access to internal structures of the base level system. This code analyses the presence and the usage of classes which implement concepts defined in the knowledge base. By executing the base level system, the aspect code tracks down information about known implementation. After collecting all the sufficient information, a metamodel (concept representation) is built.

Figure 4 shows basic architecture of the described metasystem. The base level system is weaved with an aspect of dynamic system analysis. This aspect analyses execution of the base level system. After the recognition of a predefined concept implementation, a corresponding metamodel is created. The metamodel represents a known state of the specified concept. As mentioned above, a crucial property is the fact that because of the known way of the implementation of the monitored concept it is possible to build a metamodel at a higher level of abstraction. The metamodel presented to the user contains only information related to the specified concept (at the problem domain level) and no details about the implementation.

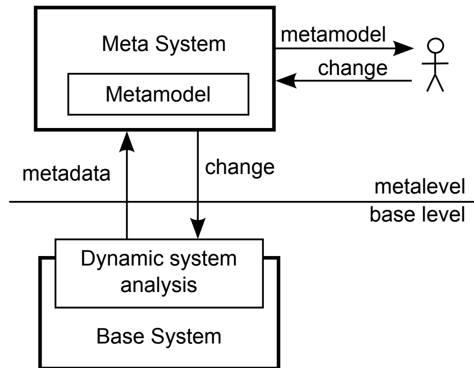

Fig. 4. A dynamic system analysis aspect is weaved into the base level system. This aspect analyzes the base level system and creates a metamodel at the metalevel. The metamodel is presented to the developer. After the metamodel change, this is reflected in the base level system modification.

3.2 A Metamodel Change

Another important aspect of the method presented is application of a change. The metasystem represents a barrier between implementation details and a metamodel at a higher level of abstraction. Just like the knowledge base defines details about concept implementation, concept model and mapping between these two, it can also define possibilities for a change. This definition must contain information about possible metamodel changes and about the way of related implementation modifications. In this way all changes supported by the metamodel are automatically propagated back to the base level system (Figures 4, 5). The base level system modification is handled by aspect-oriented programming. For example, an *around* advice is used to replace an existing method call with a new one (the original one is simply not invoked). Although this way it is not possible to apply any change of the base level system, for a certain type of changes it is sufficient.

The aspect of the base level system and the metamodel must be causally connected – the metamodel must always represent the real state of the modelled concept

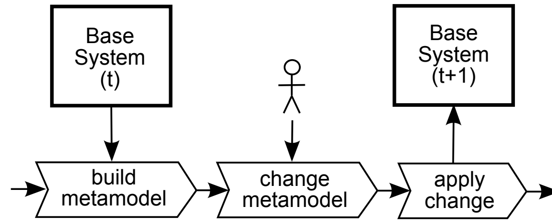

Fig. 5. The metamodel and the base level system change. A metamodel is built from a current state of the base level system in time t. After the metamodel change, the change is applied also in the base level system (in time $t + 1$).

of the base level system. All possible metamodel changes must also be immediately reflected in the base level system.

4 EXPERIMENTAL TOOL

This section describes two applications developed. The first one represents the base level system (a legacy application). The second one represents a metasystem – an experimental tool, which demonstrates possibilities and usability of the method presented. Both applications are built in Java programming language (JDK1.6.0_25). This language was chosen because of its wide use and its rich standard library (Java Class Library). As for aspect-oriented extension, AspectJ 1.6.12 was used.

The aspects play a crucial role in the experimental tool. As noted in the previous sections, the base level system is extended with a code which analyses the presence and the usage of classes which implement predefined concepts. In the described experimental tool, there is one aspect defined for each supported concept. With the help of proper *pointcut* definitions, the aspect gets all necessary information about the concept implementation. This way it is possible to get information about a concept implementation at a very fine-grained level and to distinguish different concepts (even if they are implemented by the same classes).

The following subsections describe the base level application and two predefined concepts.

4.1 The Base Level Application

The base level system is represented by a simple Address Book application. The application user may create new contacts and edit or remove existing ones. There is a possibility to import contacts from an external system (communicating over the network).

4.2 Properties of the Base Level Application

In the Java programming language, properties of an application are represented by a class named *java.util.Properties*. Each property is represented by a key-value pair. This concept can be easily modelled as a table with two columns – the key column and the value column. Each row of the table represents a single application property.

The following code snippet depicts an aspect tracking implementation of an application properties concept.

```
pointcut propertiesLoad(java.util.Properties p) :
    call(* java.util.Properties.load(..)) && target(p);
after(java.util.Properties p) returning: propertiesLoad(p) {
    ModelManager.getInstance().propertiesLoaded(p);
}
```

The experimental tool displays this concept as a table (Figure 6). It is possible to change and save values – the change is automatically reflected in the base level system. Since *java.util.Properties* class is thread-safe, changing application properties is a trivial task – it is sufficient to invoke a method *setProperty* on the class instance referenced from the metamodel.

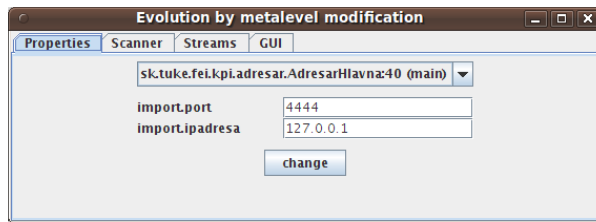

Fig. 6. Application properties change as presented by the experimental tool

Figure 7 depicts UML (Unified Modeling Language) sequential diagram of reading the base level application property. The diagram was generated by Eclipse TPTP (Test & Performance Tools Platform) and then adjusted (interactions with unimportant classes were removed to reduce size of the diagram). It is evident that the presentation of the concept as an abstraction at a problem domain level is much clearer than the sequence diagram at the implementation level.

4.3 Graphical User Interface

The second example of a concept presents an abstraction of graphical user interface dialogs. It is possible to describe dialogs of the graphical user interface as a state diagram. Each state of the diagram represents a dialog while transitions between states represent (user) actions causing activating or deactivating the dialogs

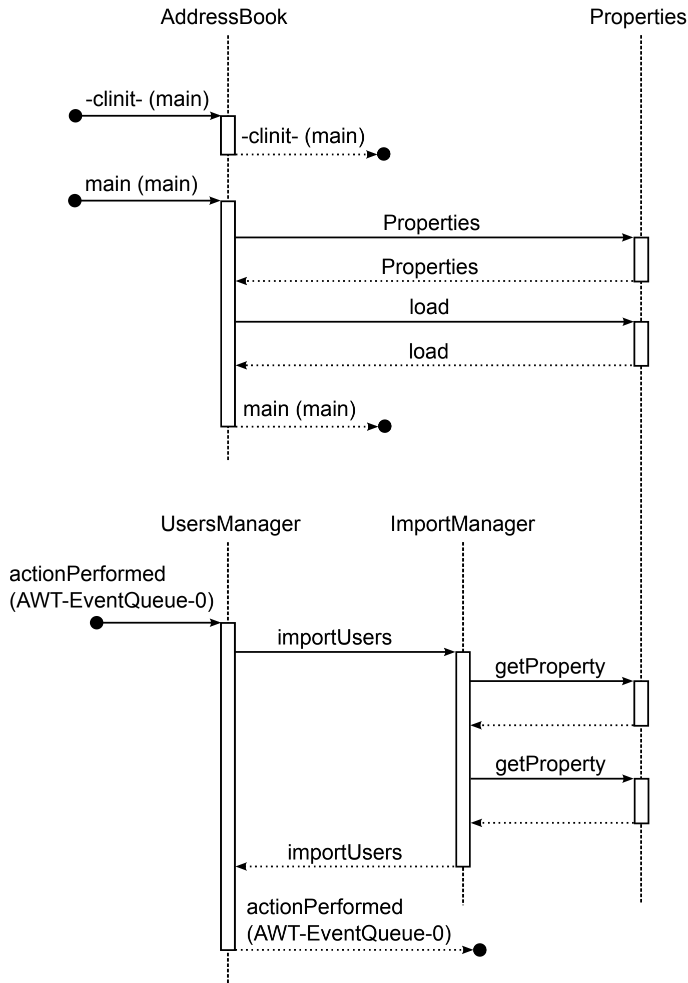

Fig. 7. Sequence diagram of activities related to the base level application properties. The first part depicts loading of properties, the second one reading a property during an action of users import.

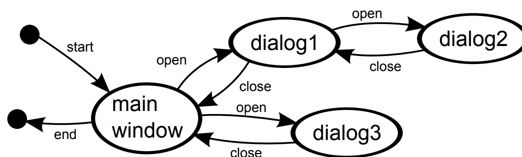

Fig. 8. A metamodel of a graphical user interface

(Figure 8). As the user works with an application, different dialogs are opening (focusing) or closing.

The aspect tracking the state of the graphical user interface is much more complicated than the one from the example of application properties and hence it is not presented here. In contrast to the previous example, it is required to track down several classes, their methods and the context in which the methods are invoked. As a result, the experimental tool automatically creates a state diagram of the used base level application's dialogs (Figure 9). A change in this state diagram (removing of a specified state) is automatically reflected as a modification in the base level application – in this case an element opening the diagram state is removed.

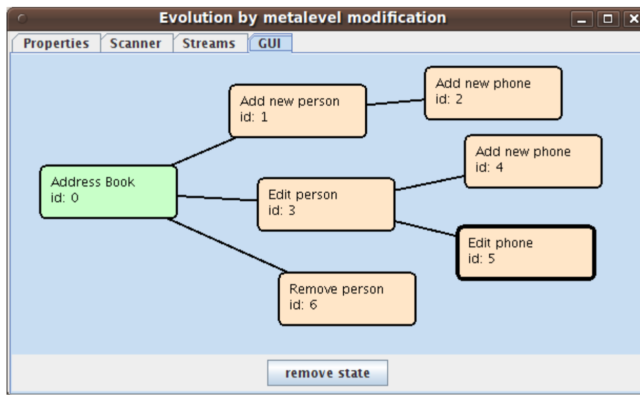

Fig. 9. A metamodel of a graphical user interface as presented by the experimental tool

In this case the resulting metamodel is also very clear and focuses on the modelled concept only. All the implementation details are hidden behind the metamodel abstraction.

5 RELATED WORK

There are many application areas, in which abstractions to metamodels may be useful: for representation of spatial data [14] by metamodels, for systems refactoring [17], for generalisation of forms in database systems [23]. Several authors utilized metalevel architectures to improve possibilities of a system evolution [7, 1, 39]. However, these works expect a programmer to build a new system following the specific rules. Such approaches are inappropriate in evolution of existing systems.

The work [31] proposes a programme of research leading to a software engineering environment which will maintain different views of the same system. These views include low-level code views and high-level architectural views. Reflection will be used to ensure that different views of the system are synchronised. Creating a soft-

ware system will involve manipulating these views. Our approach defines two different still consistent levels of the system – an implementation level and a concept level.

Using aspect-oriented programming to visualize different aspects of a subject software system is not a new idea. Paper [25] summarizes experiences with the development of a reverse engineering tool for UML sequence diagrams. Authors discuss technologies for retrieving information from Java software systems with the purpose of generating instances of a metamodel for UML sequence diagrams. They find aspect-oriented approach as the best solution – because of its elegance and non-intrusiveness of the load-time weaving mechanism in combination with the low performance impact and the expressiveness and flexibility of the join-point-based filter mechanism. They note that a tool supporting the reconstruction of the behavior of a running software system must address the major areas of data collection, representation of this data in a suitable metamodel, and finally its graphical representation.

Other approaches using aspect-oriented programming to generate UML sequence diagrams are described in [4, 15]. Aspect-oriented programming is used to instrument subject system with a tracing code; this code yields interesting information for the purpose of visualization.

As depicted in Section 4, sequence diagrams (but also class diagrams or object diagrams) are visualizing information close to the implementation level. This often results in overloading users with irrelevant low-level details. The approach described in this paper visualizes concept in a way which is close to the system's problem domain.

Paper [13] analyzes several trace exploration tools. In general, the result produced by these tools are variants of UML sequence diagrams, diagrams of interconnections between objects or the sequences of method calls. Authors conclude that the key aspect of reverse engineering is to extract different levels of abstraction of a software system. The analyzed tools visualize the content of an execution trace at some of the following levels of abstraction: statement level (the execution of every single statement of the code); object level (method interactions among objects); class level (objects of the same class are substituted with the name of their classes) and architectural level (grouping classes into clusters). From the program understanding point of view the last two levels are the most useful; but also these approaches, though providing a slightly higher level of abstraction, are still working at the level which is very close to the implementation level and far from application domain level.

Several works experiment with their own specific way of the system behavior visualization [10, 12]. The presented approaches focus mostly on the problem of huge amounts of trace data that are collected and need to be analyzed.

To use any of the described tools, the user must understand visualization output which is closer to the implementation level than to the application domain level. We see the problem in a general approach of all those methods. It is impossible to have one simple general visualization suitable for different kinds of specific behaviours. As stated in [32, 33], understanding the software behavior is a unique problem requiring a specialized solution and a visualization.

Our approach utilizes aspect-oriented programming also for applying a change in the base level application. The idea of a program modification utilizing AOP is not new; it was used already in [9, 3, 42].

6 CONCLUSIONS

In the paper we have depicted a method of concept metamodel creation. The method utilizes aspect-oriented programming to instrument the base level software system with a new code. This code analyzes the executed system and automatically builds a metamodel of a recognized predefined concept. There is also a support for predefined metamodel changes – these are automatically reflected in the base level system implementation. The base level system and the metamodel must be causally connected – the metamodel must always represent a real state of the modelled concept of the base level system, and vice versa – the metamodel change must be reflected in the base level system.

The method proposed was confirmed by an experiment. The experiment tool is able to automatically create a metamodel of a specified predefined concept of the base level system. The metamodel is created at a higher level of abstraction than the base level implementation. In the paper two predefined concepts were described – the concept of application properties and the concept of graphical user interface. These concepts were graphically presented to the user. It is more simple to understand this metamodel in comparison with other visualization techniques (as for example sequence diagrams). By changing the metamodel, it is possible to apply a change to the base level application without a need to understand all details at the implementation level.

The tool was built in Java programming language with AspectJ extension. This AOP implementation allows to weave aspects also to classes without source code – thus it is also possible to analyze and modify applications where no source code is available. The method also allows applying changes during a system runtime – there is no need to stop the base level application. All changes were implemented only by changes in the base level application object model.

The disadvantage of the method is evident – its quality heavily depends on the size and the quality of predefined concepts (defined in the knowledge base). The difficulty of concepts definitions vary – some are trivial, some complicated. The difficulty of specification of the projection between the implementation level and the metamodel at a higher level of abstraction depends on a specific concept of the base level application. The advantage is that once built knowledge base can be reused with all applications built with the same (or compatible) environment version. The process of building a complex software system consists of many repeatable parts, even when it is built using general-purpose programming language. All these parts could be the subject of the method proposed – it is not limited only to usage of the standard library.

As the next step, we find essential to propose a general way to define a concept, its implementation and a mapping between these. In the current solution, the heavy work is handled by the aspect definition – there is an analysis as well as a metamodel construction. We suppose that a metamodel format can be generalized – it seems that there is only a limited amount of required different formats of metamodels. The promising way for this generalisation – the concept model, implementation and the mapping – is designing own domain specific language (DSL) for this purpose [26, 11, 38, 40].

After designing a more general way of a concept/implementation/mapping definition, we will build a basic knowledge base. Beside library-related concepts, there are other areas suitable for concept definition (e.g. webservices, web applications, security). The analysis of other suitable areas is also a candidate for a future research.

When the knowledge base will be filled with a certain amount of different concepts, it will be desirable to test the tool with different available software projects to prove its usability.

Acknowledgement

This work was supported by VEGA Grant No. 1/0305/11 "Co-evolution of the artifacts written in domain-specific languages driven by language evolution" and by Project SK-SI-0003-10 of Slovak-Slovenian Science and Technology Cooperation "Language Patterns in Domain-Specific Language Evolution".

REFERENCES

[1] Arcelli, F.—Raibulet, C.: Evolution of an Adaptive Middleware Exploiting Architectural Reflection. In: Proceedings of ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, Nantes, France 2006.

[2] Biggerstaff, T. J.—Mitbander, B. G.—Webster, D.: The Concept Assignment Problem in Program Understanding. In: Proceedings of the 15th international conference on Software Engineering – ICSE '93, Los Alamitos, CA, USA, IEEE Computer Society Press 1993, pp. 482–498.

[3] Bluemke, I.—Billewicz, K.: Aspect Modification of an EAR Application. CIS2E '08, Krakow, Poland, Springer 2008.

[4] Briand, L. C.—Labiche, Y.—Leduc, J.: Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. IEEE Trans. Softw. Eng., Vol. 32, pp. 642–663, September 2006.

[5] Brooks, R.: Using a Behavioral Theory of Program Comprehension in Software Engineering. In: Proceedings of the 3rd International Conference on Software Engineering – ICSE '78, Piscataway, NJ, USA, IEEE Press 1978, pp. 196–201.

[6] Brooks, Jr.—Frederick, P.: No Silver Bullet Essence and Accidents of Software Engineering. Computer, Vol. 20, 1987, No. 4, pp. 10–19I, EEE Computer Society Press, Los Alamitos, CA, USA, ISSN 0018-9162.

[7] Cazzola, W.—Sosio, A.—Tisato, F.: Shifting Up Reflection from the Implementation to the Analysis Level. In: Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999, Springer-Verlag, London, UK, 2000, ISBN 3-540-67761-5, pp. 1–20.

[8] Chen, K.—Rajlich, V.: Case Study of Feature Location Using Dependence Graph. In: Proceedings of the 8th International Workshop on Program Comprehension – IWPC '00, IEEE Computer Society, Washington, DC, USA, 2000, p. 241.

[9] Cheng, L. T.—Patterson, J.—Rohall, S. L.—Hupfer, S.—Ross, S.: Weaving a Social Fabric into Existing Software. In: Proceedings of the 5th International conference on Aspect-oriented software development – AOSD 05, Chicago, USA 2005, pp. 147–159.

[10] Cornelissen, B.—Zaidman, A.—Holten, D.—Moonen, L.—van Deursen, A.—van Wijk, J. J.: Execution Trace Analysis Through Massive Sequence and Circular Bundle Views. Journal of Systems and Software, Vol. 81, 2008, No. 12, pp. 2252–2268.

[11] Črepinšek, M.—Mernik, M.—Bryant, B.—Javed, F.—Sprague, F.: Inferring Context-Free Grammars for Domain-Specific Languages. Electronic notes in theoretical computer science, Vol. 141, 2005, No. 4, pp. 99–116.

[12] Greevy, O.—Lanza, M.—Wysseier, C.: Visualizing Live Software Systems in 3D. In: Proceedings of the 2006 ACM Symposium on Software Visualization – SoftVis '06, ACM, New York, NY, USA, pp. 47–56.

[13] Hamou-Lhadj, A.—Lethbridge, T. C.: A Survey of Trace Exploration Tools and Techniques. In: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative research – CASCON '04, IBM Press 2004, pp. 42–55.

[14] Dlugolinský, Š.—Laclavík, M.—Hluchý, L.: Towards a Search System for the Web Exploiting Spatial Data of a Web Document. In: Proceedings of Database and Expert Systems Applications – DEXA 2010, R. R. Wagner (Ed.), Los Alamitos, IEEE Computer Society 2010, pp. 27–31.

[15] Khaled, R.—Noble, J.—Biddle, R.: InspectJ: Program Monitoring for Visualisation Using AspectJ. In: Proceedings of the 26th Australasian Computer Science Conference – ACSC '03, Australian Computer Society, Inc., Darlinghurst, Australia 2003, Vol. 16, pp. 359–368.

[16] Kiczales, G.—Lamping, J.—Mendhekar, A.—Maeda, C.—Lopes, C. V.—Loingtier, J. M.—Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming – ECOOP 1997, Jyväskylä, Finland, Vol. 1241 of Lecture Notes in Computer Science, pp. 220–242.

[17] Kitlei, R.—Lövei, L.—Nagy, T.—Horváth, Z.—Kozsik, T.: Layout Preserving Parser for Refactoring in Erlang. Acta Electrotechnica et Informatica, Vol. 9, 2009, No. 3, pp. 54–63, ISSN 1335–8243.

[18] LEHMAN, M. M.: Laws of Software Evolution Revisited. In: Proceedings of the 5th European Workshop on Software Process Technology – EWSPT '96, Springer-Verlag London, United Kingdom 1996, pp. 108–124, LNCS 1149.

[19] LEHMAN, M. M.—RAMIL, J. F.: An Approach to a Theory of Software Evolution. In: Proceedings 4th International Workshop on Principles of Software Evolution – IWPSE '01, ACM Press, NY, USA 2001, pp. 70–74.

[20] LEHMAN, M. M.—RAMIL, J. F.—KAHEN, G.: A Paradigm for the Behavioural Modelling of Software Processes using System Dynamics. Technical Report 2001/8, Department of Computing, Imperial College, London, United Kingdom, September 2001, pp. 1–11.

[21] LEHMAN, M. M.—RAMIL, J. F.—WERNICK, P. D.—PERRY, D. E.—TURSKI, W. M.: Metrics and Laws of Software Evolution – The Nineties Views. In: Proceedings of the 4th International Symposium on Software Metrics – METRICS '97, IEEE Computer Society Press, Washington, DC, USA 1997, pp. 20–32.

[22] LIU, D.—MARCUS, A.—POSHYVANYK, D.—RAJLICH, V.: Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering – ASE '07, ACM, New York, NY, USA 2007, pp. 234–243, ISBN 978-1-59593-882-4.

[23] LUKOVIČ, I—MOGIN, P.—PAVIČEVIČ, J.—RISTIČ, S: An Approach to Developing Complex Database Schemas Using Form Types. Software Practice & Experience, Vol. 37, 2007, No. 15, pp. 1621–1656.

[24] MARCUS, A.—SERGEYEV, A.—RAJLICH, V.—MALETIC, J. I.: An Information Retrieval Approach to Concept Location in Source Code. In: Proceedings of the 11th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA 2004, pp. 214–223, ISBN 0-7695-2243-2.

[25] MERDES, M.—DORSCH, D.: Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development. In: Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java – PPPJ '06, ACM, New York, NY, USA 2006, pp. 125–134.

[26] MERNIK, M.—HEERING, J.—SLOANE, A. M.: When and How to Develop Domain-Specific Languages. ACM Comput. Surv., Vol. 37, 2005, No. 4, pp. 316–344.

[27] OLSZAK, A.—JÖRGENSEN, B. N.: Remodularizing Java Programs for Comprehension of Features. In: Proceedings of the First International Workshop on Feature-Oriented Software Development – FOSD '09, ACM, New York, NY, USA 2009, pp. 19–26.

[28] ORIOL, M.—CAZZOLA, W.—CHIBA, S.—SAAKE, G.: Getting Farther on Software Evolution via AOP and Reflection. Report on the 5th RAM-SE Workshop at ECOOP 2008, RAM-SE '08, Springer-Verlag, Berlin, Heidelberg 2009, pp. 63–69, ISBN 978-3-642-02046-9.

[29] PETRENKO, M.—RAJLICH, V.—VANCIU, R.: Partial Domain Comprehension in Software Evolution and Maintenance. In: Proceedings of the 16th IEEE International Conference on Program Comprehension – ICPC '08, IEEE Computer Society, Washington, DC, USA 2008, pp. 13–22, ISBN 978-0-7695-3176-2.

[30] Rajlich, V.: Intensions Are a Key to Program Comprehension. In: IEEE 17th International Conference on Program Comprehension, ICPC '09, May 2009, pp. 1–9, ISSN 1063-6897.

[31] Rank, S.: Architectural Reflection for Software Evolution. In: Proceedings of the 2nd Workshop on Reflection, AOP and Meta-Data for Software Evolution – ECOOP 2005, Scotland.

[32] Reiss, S. P.: Visualizing Program Execution Using User Abstractions. In: Proceedings of the 2006 ACM Symposium on Software Visualization – SoftVis '06, ACM, New York, NY, USA 2006, pp. 125–134.

[33] Reiss, S. P.: Visual Representations of Executing Programs. Journal of Visual LanAOP Approach to Metamodel Abstraction Languages and Computing, Vol. 18, 2007, No. 2, pp. 126–148, ISSN 1045-926X.

[34] Roethlisberger, D.—Greevy, O.—Nierstrasz, O.: Feature Driven Browsing. In: Proceedings of the 2007 international Conference on Dynamic languages, in conjunction with the 15th International Smalltalk Joint Conference 2007 – ICDL '07, ACM, New York, NY, USA 2007, pp. 79–100, ISBN 978-1-60558-084-5.

[35] Rugaber, S.: Program Comprehension. A. Kent and J. G. Williams (Eds): Encyclopedia of Computer Science and Technology, April 1995, pp. 341–368.

[36] van Sickle, L.—Hartman, J.: Technical Introduction to the First Workshop on Artificial Intelligence and Automated Program Understanding. 1992, pp. 12–16.

[37] Wilde, N.—Scully, M. C.: Software Reconnaissance: Mapping Program Features to Code. Journal of Software Maintenance, Vol. 7, 1995, No. 1, pp. 49–62, ISSN 1040-550X.

[38] Porubän, J.—Václavík, P.: Extensible Language Independent Source Code Refactoring. In: AEI 2008: International Conference on Applied Electrical Engineering and Informatics, Greece, Athens, September 8–11. Košice: FEI TU, 2008, pp. 58–63.

[39] Porubän, J.—Sabo, M.: Jessine: Integrating Rules in Enterprise Software Applications. Journal of Information, Control and Management Systems, Vol. 7, 2009, No. 1, pp. 81–88.

[40] Porubän, J.—Sabo, M.: Preserving Design Patterns Using Source Code Annotations. Journal of Computer Science and Control Systems, Vol. 2, 2009, No. 1, pp. 53–56.

[41] Václavík, P.: Application Domain Name-Based Analysis. Journal of Computer Science and Control Systems, Vol. 2, 2009, No. 2, pp. 66–69.

[42] Vranič, V.—Menkyna, R.—Bebjak, M.—Dolog, P.: Aspect-Oriented Change Realizations and Their Interaction. e-Informatica Software Engineering Journal, Vol. 3, 2009, No. 1, pp. 43–58.

Ján Kollár is Full Professor of Informatics at Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his M. Sc. summa cum laude in 1978 and his Ph. D. in computer science in 1991. In 1978–1981 he was with the Institute of Electrical Machines in Košice. In 1982–1991 he was with Institute of Computer Science at the P. J. Šafárik University in Košice. Since 1992 he is with the Department of Computer and Informatics at the Technical University of Košice. In 1985 he spent 3 months at the Joint Institute of Nuclear Research in Dubna, USSR. In 1990 he spent 2 months at the Department of Computer Science at Reading University, UK. He was involved in research projects dealing with real-time systems, the design of microprogramming languages, image processing and remote sensing, dataflow systems, implementation of programming languages, and high performance computing. He is the author of process functional programming paradigm. Currently his research area covers formal languages and automata, programming paradigms, implementation of programming languages, functional programming, and adaptive software and language evolution.

Michal Vagaš is Assistant Professor of Informatics at Department of Computers and Informatics, Matej Bel University, Banská Bystrica, Slovakia. He received his M. Sc. in 2001. The subject of his research imcludes meta-programming, programming paradigms, aspect-oriented programming and system evolution.